duat Documentation

Release 0.2.0

Dih5

Jun 27, 2018

Contents

CHAPTER 1

Introductory demo

This document is a functional Jupyter notebook showing how to work with OSIRIS using the duat Python interface. You can also use it with python scripts or from the interpreter, but the notebook is a handy way to use the package I suggest you to consider.

The updated version of this notebook can be download here.

First, import some modules.

```
In [1]: import os
        import numpy as np
        import matplotlib.pyplot as plt
```

```
In [2]: from duat import config, plot, run
```

```
/usr/lib/python3.6/site-packages/h5py/__init__.py:36: FutureWarning: Conversion of the second argumer
  from ._conv import register_converters as _register_converters
```

If a warning on not found executables was raised for you, add some code setting the path to the folder with the executables: `run.set_osiris_path(path.join("path", "to", "osiris", "folder"))`

1.1 Creating a simulation

The config module offers functionality to create a simulation.

```
In [3]: # Create a config file with the defaults
        sim = config.ConfigFile(1)  # Argument -> dimension
        # Check the generated code
        print(sim)
```

```
node_conf
{
  node_number(1:1) = 1,
  if_periodic(1:1) = .true.,
}

grid
```

```
{
  coordinates = "cartesian",
  nx_p(1:1) = 1024,
}

time_step
{
  dt = 0.07,
  ndump = 10,
}

space
{
  xmin(1:1) = 0,
  xmax(1:1) = 102.4,
  if_move(1:1) = .false.,
}

time
{
  tmin = 0,
  tmax = 7,
}

emf_bound
{
  type(1:2, 1) = 0, 0,

}

particles
{
  num_species = 2,
  num_cathode = 0,
  num_neutral = 0,
  num_neutral_mov_ions = 0,
}

!---Species configuration
!---Configuration for species 1
species
{
  num_par_max = 2048,
  rqm = -1,
  num_par_x(1:1) = 2,
  vth(1:3) = 0, 0, 0,
  vfl(1:3) = 0, 0, 0,
}

profile
{
  fx(1:6, 1) = 1, 1, 1, 1, 1, 1,

  x(1:6, 1) = 0, 0.9999, 1, 2, 2.001, 10000,

}

spe_bound
{
```

```
  type(1:2, 1) = 0, 0,

}

diag_species
{
}


!---Configuration for species 2
species
{
  num_par_max = 2048,
  rqm = -1,
  num_par_x(1:1) = 2,
  vth(1:3) = 0, 0, 0,
  vfl(1:3) = 0, 0, 0,
}

profile
{
  fx(1:6, 1) = 1, 1, 1, 1, 1, 1,

  x(1:6, 1) = 0, 0.9999, 1, 2, 2.001, 10000,

}

spe_bound
{
  type(1:2, 1) = 0, 0,

}

diag_species
{
}
```

```
In [4]: # Let's change some parameters:
        # Parameters can be edited using the python item access notation
        sim["time"]["tmax"] = 30.0
        # Beware the python indexes starting at zero. E.g., "1" is the second particle species
        sim["species_list"][0]["species"]["vfl"] = [0.0, 0.0, 0.6]
        sim["species_list"][0]["species"]["num_par_x"] = [200]
        sim["species_list"][1]["species"]["vfl"] = [0.0, 0.0, -0.6]
        sim["species_list"][1]["species"]["num_par_x"] = [200]

        # Order of output is handled by duat. We can now change parameters that will appear before i
        sim["species_list"][0]["diag_species"].set_pars(ndump_fac=1, reports="ene")

        # We can benefit from python preprocessing power
        ene_bins = np.arange(0, 0.5, 0.02)
        sim["species_list"][1]["diag_species"].set_pars(ndump_fac=1, ndump_fac_pha=1, pha_ene_bin="x1
                                                        ene_bins=ene_bins, n_ene_bins=len(ene_bins))

In [5]: # Even if a section was not created before, accessing it with the index notation will create
        sim["diag_emf"]["reports"] = ["e1", "e2", "e3"]
```

```
        sim["diag_emf"]["ndump_fac"] = 5

In [6]: # This also works with lists
        sim["zpulse_list"][0].set_pars(a0=1.0, omega0=1.0, phase=0.0, pol_type=1, pol=0, propagation=
                                       lon_type="gaussian", lon_x0=40, lon_range=20, lon_duration=50,

In [7]: # Check the generated code again
        print(sim)
```

```
node_conf
{
  node_number(1:1) = 1,
  if_periodic(1:1) = .true.,
}

grid
{
  coordinates = "cartesian",
  nx_p(1:1) = 1024,
}

time_step
{
  dt = 0.07,
  ndump = 10,
}

space
{
  xmin(1:1) = 0,
  xmax(1:1) = 102.4,
  if_move(1:1) = .false.,
}

time
{
  tmin = 0,
  tmax = 30,
}

emf_bound
{
  type(1:2, 1) = 0, 0,

}

diag_emf
{
  reports(1:3) = "e1", "e2", "e3",
  ndump_fac = 5,
}

particles
{
  num_species = 2,
  num_cathode = 0,
  num_neutral = 0,
  num_neutral_mov_ions = 0,
}
```

```
!---Species configuration
!---Configuration for species 1
species
{
  num_par_max = 2048,
  rqm = -1,
  num_par_x(1:1) = 200,
  vth(1:3) = 0, 0, 0,
  vfl(1:3) = 0, 0, 0.6,
}

profile
{
  fx(1:6, 1) = 1, 1, 1, 1, 1, 1,

  x(1:6, 1) = 0, 0.9999, 1, 2, 2.001, 10000,

}

spe_bound
{
  type(1:2, 1) = 0, 0,

}

diag_species
{
  ndump_fac = 1,
  reports = "ene",
}


!---Configuration for species 2
species
{
  num_par_max = 2048,
  rqm = -1,
  num_par_x(1:1) = 200,
  vth(1:3) = 0, 0, 0,
  vfl(1:3) = 0, 0, -0.6,
}

profile
{
  fx(1:6, 1) = 1, 1, 1, 1, 1, 1,

  x(1:6, 1) = 0, 0.9999, 1, 2, 2.001, 10000,

}

spe_bound
{
  type(1:2, 1) = 0, 0,

}

diag_species
{
  ndump_fac = 1,
```

```
  ndump_fac_pha = 1,
  pha_ene_bin = "x1_|charge|",
  ene_bins(1:25) = 0, 0.02, 0.04, 0.06, 0.08, 0.1, 0.12, 0.14, 0.16, 0.18, 0.2, 0.22, 0.24, 0.26, 0.2
  n_ene_bins = 25,
}



!---zpulse_list
zpulse
{
  a0 = 1,
  omega0 = 1,
  phase = 0,
  pol_type = 1,
  pol = 0,
  propagation = "forward",
  lon_type = "gaussian",
  lon_x0 = 40,
  lon_range = 20,
  lon_duration = 50,
  per_type = "plane",
}
```

1.2 Running a simulation

The config module offers functionality to create a simulation

```
In [8]: # A directory for the run
        run_dir = os.path.join(os.path.expanduser("~"), "test-run")

In [9]: # Run the simulation. This also checks for errors, but note they are not always detected
        myrun = run.run_config(sim, run_dir, clean_dir=True)

In [10]: # The created run instance offers some information of the run in progress.
         myrun

Out[10]: Run<test-run [RUNNING (160/429)]>

In [11]: # Other methods are available in the Run object. See the documentation for more information
         print("Size in disk of the run: %.2f MiB" % (myrun.get_size()/1024/1024))

Size in disk of the run: 3.78 MiB
```

1.3 Plotting the results

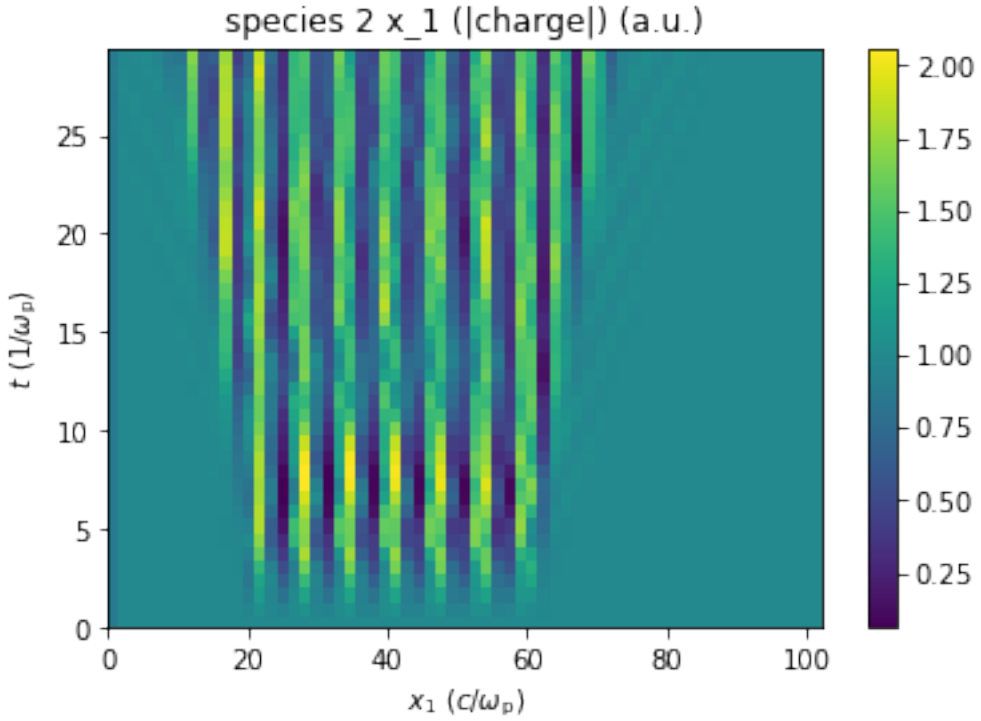
The plot module offers functionality to create a simulation

```
In [12]: # Diagnostic objects can be used even if the run is in progress
         diagnostics = myrun.get_diagnostic_list()
         for d in diagnostics:
             print(d)

Diagnostic<species 2 x_1 (|charge|) ([64], 26, 43)>
Diagnostic<E_2 ([1024], 1, 9)>
```

```
Diagnostic<E_1 ([1024], 1, 9)>
Diagnostic<E_3 ([1024], 1, 9)>
Diagnostic<Kinetic Energy ([1024], 1, 43)>
```

```
In [13]: # Methods like time_1d_colormap allow to represent a map
         fig, ax = diagnostics[0].time_1d_colormap(dataset_selector=np.sum)
         # The returned Figure and Axes instances allow for customization and export.
         # Note the method itself takes a parameter for automatic exportation
```

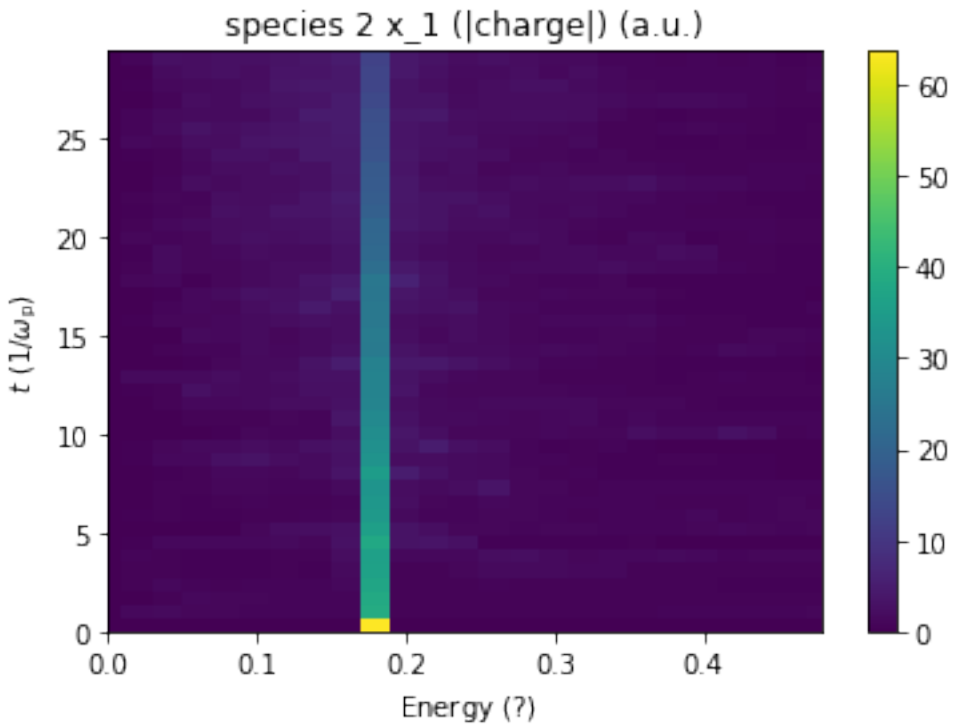



```
In [14]: # The time_1d_animation method allows to visualize a function in time
         fig, ax, anim = diagnostics[2].time_1d_animation()
         # This can be exported using the anim object or given a parameter to the function.

In [15]: # In Jupyter you can plot this with:
         from IPython.display import HTML
         HTML(anim.to_html5_video())
         # mpeg must be installed for this to work!
         # The video can be downloaded from the output.

Out[15]: <IPython.core.display.HTML object>

In [16]: # Do the same as a color map
         fig, ax = diagnostics[0].time_1d_colormap(axes_selector=(np.sum,))
```




```
In [17]: # For manual manipulation use the get_generator method
         gen = diagnostics[1].get_generator()
         # This returns a generator that provides data when iterated
         for snapshot in gen:
             print(snapshot)
```

```
[0. 0. 0. ... 0. 0. 0.]
[0. 0. 0. ... 0. 0. 0.]
[0. 0. 0. ... 0. 0. 0.]
[0. 0. 0. ... 0. 0. 0.]
[ 0.  0.  0. ... -0.  0.  0.]
[1.1072185e-14 5.5117166e-14 2.6307391e-13 ... 6.9920492e-17 3.9428322e-16
 2.1332491e-15]
[-0.00111232 -0.01204932 -0.02006297 ...  0.0090847   0.01041243
  0.00722775]
[ 0.01315147  0.01790592  0.0081597  ...  0.00850101 -0.00061765
 -0.00045989]
[0.04846063 0.06358237 0.05960049 ... 0.01152259 0.02055315 0.02533524]
```

CHAPTER 2

Cookbook

2.1 Create a OSIRIS config file

```
from duat import config
sim = config.ConfigFile(1)  # 1D
# Use index notation to modify or create parameters and sections
sim["grid"]["nx_p"] = 4000
# Check the generated fortran code
print(sim)
```

Sections are almost like regular OSIRIS input, although repeatable sections have a `_list` suffix; for example, `sim["species_list"][0]` is the first species. See structure for details.

2.2 Run a config file

```
from duat import run
# [...]
# Create a ConfigFile instance sim
myrun = run.run_config(sim, "/home/user/myrun", blocking=False, clean_dir=True)
# The returned Run instance offers some useful info. Check the API documentation
```

To run in a grid system use `run_config_grid` instead.

2.3 Run a variation

```
from duat import config, run
# [...]
# Create a ConfigFile instance sim
parameter = ["zpulse_list", 0, "a0"] # Parameter to change
```

(continues on next page)

(continued from previous page)

```
values = [0.2, 0.5, 0.7, 1.0, 1.2] # Values to take
var = config.Variation((parameter, values)) # Add more 2-tuples for cartesian product␣
↪of parameter variation
run_list = run.run_variation(sim, var, "/home/user/myvar", caller=3) # Create three␣
↪threads executing simulations
```

To run in a grid system use `run_variation_grid` instead.

2.4 Plot results with matplotlib

```
# [...]
# Obtain a Run instance named myrun
# Diagnostic objects can be used even if the run is in progress
diagnostics = myrun.get_diagnostic_list()
for d in diagnostics:
    print(d)
# Suppose diagnostics[0] is 1D. Then:
fig, ax = diagnostics[0].time_1d_colormap()
# Now export, show, customize or whatever
```

CHAPTER 3

ConfigFile structure

By running

```
from duat import config
print(config.ConfigFile.get_structure())
```

you can see an updated description of how an OSIRIS configuration file is described in duat. The output is the following:

- simulation (Section)
- node_conf (Section)
- grid (Section)
- time_step (Section)
- restart (Section)
- space (Section)
- time (Section)
- el_mag_fld (Section)
- emf_bound (Section)
- smooth (Section)
- diag_emf (Section)
- particles (Section)
- species_list (SpeciesList)
  - 0 (Species)
    - species (Section)
    - profile (Section)
    - spe_bound (Section)

        - diag_species (Section)
    - 1 . . .
- cathode_list (SpeciesList)
    - 0 (Species)
        - species (Section)
        - profile (Section)
        - spe_bound (Section)
        - diag_species (Section)
    - 1 . . .
- neutral_list (NeutralList)
    - 0 (Neutral)
        - neutral (Section)
        - profile (Section)
        - diag_neutral (Section)
        - species (Section)
        - spe_bound (Section)
        - diag_species (Section)
    - 1 . . .
- neutral_mov_ions_list (NeutralMovIonsList)
    - 0 (Section)
    - 1 . . .
- collisions (Section)
- zpulse_list (ZpulseList)
    - 0 (Section)
    - 1 . . .
- current (Section)
- smooth_current (SmoothCurrent)

CHAPTER 4

API reference

4.1 duat.config

Model OSIRIS configuration files.

class duat.config.**Cathode**(*label=None*)
Set of sections defining a cathode.

__init__(*label=None*)
Initialize self. See help(type(self)) for accurate signature.

class duat.config.**CathodeList**(*label='cathodes'*)
List of sections defining the cathodes

__init__(*label='cathodes'*)
Args: label (str): a label inserted as a comment when generating fortran code. lst (list of MetaSection): a preexisting list of sections conforming this one. default_type (str or class): the default type for implicit creation of content. If a str, a Section with the

provided name will be created. Otherwise, a instance of the provided class with no parameters will be created.

default_type
alias of *Cathode*

class duat.config.**ConfigFile**(*d=1*, *template='default'*)
Set of Sections defining an input file.

__init__(*d=1*, *template='default'*)
Create a default d-dimensional config file.

Parameters

- **d** (*int*) – The number of dimensions (1, 2 or 3), used if a template is provided.
- **template** (*str*) – A model for the initial configuration. Available values are: * "default": A basic configuration based on the examples. * "none": Absolutely no configuration.

clone()
Get a deep copy of the instance

Returns a deep copy of this instance

Return type *ConfigFile*

classmethod from_file(*file_path*)
Create and return an instance from a fortran code representation in a file

classmethod from_string(*string*)
Create and return an instance from a fortran code representation

get_d()
Get the dimension of the configuration file.

Returns

The dimension according to the mandatory xmax parameter in the space section. 0 if it could not be found.

Return type int

get_nodes()
Get the number of nodes required by the instance.

Returns Number of nodes.

Return type int

to_fortran()
Return a str with the fortran code representing the instance

write(*path*)
Save the config file to the specified path.

Parameters **path** – the path of the output file.

Raises OSError – if the file could not be opened.

class duat.config.**MetaSection**
Something behaving like a set of sections.

__str__()
Return str(self).

classmethod get_structure(*offset=0*)
Get a string representing the structure of the class.

This methods returns a string with lines in the format "- index (type)", where index is the one used to access a subitem and type is the class of that item.

Parameters **offset** – Two spaces times this number will be used to indent the returned string. This is used to generate a multi-level description.

Returns A representation of the structure of the class.

Return type (str)

to_fortran()
Return a str with the fortran code representing the instance

class duat.config.**Neutral**(*label=None*)
Set of sections defining a neutral.

__init__(*label=None*)
Initialize self. See help(type(self)) for accurate signature.

class duat.config.**NeutralList**(*label='neutrals'*)
List of sections defining the neutrals

__init__(*label='neutrals'*)
Args: label (str): a label inserted as a comment when generating fortran code. lst (list of MetaSection): a preexisting list of sections conforming this one. default_type (str or class): the default type for implicit creation of content. If a str, a Section with the

provided name will be created. Otherwise, a instance of the provided class with no parameters will be created.

default_type
alias of *Neutral*

class duat.config.**NeutralMovIonsList**(*label='neutral moving ions'*)
List of sections defining the neutral moving ions

__init__(*label='neutral moving ions'*)
Args: label (str): a label inserted as a comment when generating fortran code. lst (list of MetaSection): a preexisting list of sections conforming this one. default_type (str or class): the default type for implicit creation of content. If a str, a Section with the

provided name will be created. Otherwise, a instance of the provided class with no parameters will be created.

class duat.config.**Section**(*name*, *param=None*)
The class defining a configuration block.

__init__(*name*, *param=None*)
Initialize self. See help(type(self)) for accurate signature.

remove_par(*name*)
Remove the given parameter from the Section

set_par(*name*, *val*)
Add or update the value of a parameter

set_pars(***kwargs*)
Add or update some parameters using keyword arguments

to_fortran()
Return a str with the fortran code representing the instance

class duat.config.**SectionList**(*label=None*, *lst=None*, *default_type=None*)
Class defining a list of sections in a numerical order.

Here 'section' refers to any subclass of *MetaSection*.

__init__(*label=None*, *lst=None*, *default_type=None*)

Parameters

- **label** (*str*) – a label inserted as a comment when generating fortran code.
- **lst** (*list of MetaSection*) – a preexisting list of sections conforming this one.
- **default_type** (*str or class*) – the default type for implicit creation of content. If a str, a Section with the provided name will be created. Otherwise, a instance of the provided class with no parameters will be created.

default_type
alias of *Section*

classmethod get_structure(*offset=0*)
Get a string representing the structure of the class.

This methods returns a string with lines in the format "- index (type)", where index is the one used to access a subitem and type is the class of that item.

Parameters **offset** – Two spaces times this number will be used to indent the returned string. This is used to generate a multi-level description.

Returns A representation of the structure of the class.

Return type (str)

remove_section(*number=-1*)
Remove the section in the position given the index. By default, the last one.

to_fortran()
Return a str with the fortran code representing the instance

class duat.config.**SectionOrdered**(*label=None*, *order=None*, *fixed=True*, *types=None*)
Class defining a set of sections that must be outputted in a particular order given by a keyword related to them.

Here 'section' refers to any subclass of *MetaSection*.

__init__(*label=None*, *order=None*, *fixed=True*, *types=None*)
Initialize self. See help(type(self)) for accurate signature.

classmethod get_structure(*offset=0*)
Get a string representing the structure of the class.

This methods returns a string with lines in the format "- index (type)", where index is the one used to access a subitem and type is the class of that item.

Parameters **offset** – Two spaces times this number will be used to indent the returned string. This is used to generate a multi-level description.

Returns A representation of the structure of the class.

Return type (str)

remove_section(*name*)
Remove the section with the given name

set_section(*name*, *section=None*)
Add or replace a section.

to_fortran()
Return a str with the fortran code representing the instance

class duat.config.**SmoothCurrent**(*name*)

__init__(*name*)
Initialize self. See help(type(self)) for accurate signature.

class duat.config.**Species**(*label=None*, *dim=None*)
Set of sections defining a species.

__init__(*label=None*, *dim=None*)
Initialize self. See help(type(self)) for accurate signature.

class duat.config.**SpeciesList**(*label='species'*)
List of sections defining the species

__init__(*label='species'*)
Args: label (str): a label inserted as a comment when generating fortran code. lst (list of MetaSection): a preexisting list of sections conforming this one. default_type (str or class): the default type for implicit creation of content. If a str, a Section with the

provided name will be created. Otherwise, a instance of the provided class with no parameters will be created.

default_type
alias of *Species*

class duat.config.**Variation**(**args*, *epilog=None*)
Represents a variation of a set of parameters in config files.

Each parameter varies in list of values. The configuration files produced by this class take into account all combinations of values, i.e., the parameter space is given by the cartesian product.

__init__(**args*, *epilog=None*)
Create a Variation with the given parameters and values.

Parameters

- ***args** (2-tuple of list) – Each argument must be a 2-tuple whose first elements is a list of str or int which identifies the parameter in its section and a list of the values the parameter will take. The list can be None to perform no action while passing the parameter to the epilog function (see below).
- **epilog** (*callable*) – A function of two arguments that will be called with the simulation and the list of parameters when a config is being generated. This can be used for advanced modification, for example, to set two parameters to a related value (like two species temperature).

__repr__()
Return repr(self).

get_generator(*config*)
Get a generator that produces ConfigFile objects following the Variation.

Parameters **config** (ConfigFile) – The configuration where the Variation will be applied.

Returns A generator which provides the ConfigFile instances.

Return type generator

get_parameter_list()
Get a list with the parameter values in the same order that get_generator().

This method might be useful to post-process the results if the parameter space is simple.

Returns A list with the values of the parameters in the cartesian product order.

Return type list of *tuple*

class duat.config.**ZpulseList**(*label='zpulses'*)
List of sections defining the laser pulses

__init__(*label='zpulses'*)
Args: label (str): a label inserted as a comment when generating fortran code. lst (list of MetaSection): a preexisting list of sections conforming this one. default_type (str or class): the default type for implicit creation of content. If a str, a Section with the

provided name will be created. Otherwise, a instance of the provided class with no parameters will be created.

4.2 duat.run

Run configuration files with OSIRIS.

class duat.run.**Run**(*run_dir*)
An osiris run.

run_dir
str – Directory where the run takes place.

total_steps
int – Amount of time steps in the simulation.

running_mode
str – Can be "local" if a local process was found, "grid" if a grid job was found, or "" otherwise. A simulation launch in the grid but running in the local machine will be tagged as "local" (since you can access its process).

processes
list of psutil.Process – If running_mode is "local", representation of the processes running the simulation.

job
dict – If running_mode is "grid", information about the job of the simulation.

__init__(*run_dir*)
Create a Run instance.

Parameters **run_dir** (*str*) – Path where the OSIRIS run takes place. An os-stdin file must exist there.

Raises FileNotFoundError – If no os-stdin is found.

__repr__()
Return repr(self).

current_step()
Find the current simulation step.

Returns The simulation step or -1 if it could not be found.

Return type int

estimated_time()
Estimated time to end the simulation in seconds.

The estimation uses a linear model and considers initialization negligible. For local runs, the start time of a process is used in the calculation. For grid runs, the start time of the job is used instead.

If the run was resumed, the estimation will be wrong.

Returns The estimation of the time to end the simulation or NaN if no estimation could be done.

Return type float

get_config()
Return a ConfigFile instance parsing the os-stdin file

get_diagnostic_list()
Create a list with the diagnostic found in the given Run.

Returns List of the diagnostic found.

Return type `list` of `Diagnostic`

get_size()
Get the size of all run data in bytes.

get_status()
Return the status of the run.

Returns

The status of the run. Possible values are:

- "RUNNING": Either a process is running the simulation (running_mode="local") or a qstat job was found (running_mode="grid").
- "FAILED": An error was detected in the OSIRIS output.
- "FINISHED": The simulation has successfully finished its execution.
- "INCOMPLETE": The simulation appears to have started but not running now. You should check the output to try to understand what happened. If restart information was sent, you might want to run "continue.sh" to resume the simulation.
- "NOT STARTED": The files were created, but the simulation is not running. Note you will see this state if the Run is queued in a MPCaller instance.

Return type str

kill()
Abruptly terminate the OSIRIS processes (if running).

The `terminate()` method should be used instead to perform a cleaner exit.

If runnning is "local", sends SIGKILL to the processes. If "grid", calls qdel.

Raises `subprocess.CalledProcessError` – If using a grid and qdel fails.

real_time()
Find the total time in seconds taken by the simulation if it has finished, otherwise returning nan.

terminate()
Terminate the OSIRIS processes (if running).

If runnning is "local", sends SIGINT to the processes. If "grid", calls qdel.

Raises `subprocess.CalledProcessError` – If using a grid and qdel fails.

duat.run.**open_run_list**(*base_path*, *filter=None*)
Create a Run instance for each of the subdirectories in the given path.

Parameters

- **base_path** (*str*) – Path where the runs are found.
- **filter** (*str*) – Filter the directories using a UNIX-like pattern.

Returns A list with the Run instances, ordered so their paths are in human order.

Return type list of *Run*

duat.run.**osiris_1d = ''**
str – Path to the osiris-1D.e file

duat.run.**osiris_2d = ''**
str – Path to the osiris-2D.e file

duat.run.**osiris_3d = ''**
str – Path to the osiris-3D.e file

duat.run.**run_config**(*config*, *run_dir*, *prefix=None*, *clean_dir=True*, *blocking=None*, *force=None*, *mpcaller=None*, *create_only=False*)
Initiate a OSIRIS run from a config instance.

Parameters

- **config** (*ConfigFile*) – The instance describing the configuration file.
- **run_dir** (*str*) – Folder where the run is carried.
- **prefix** (*str*) – A prefix to run the command. If None, "mpirun -np X " will be used when a config with multiple nodes is provided.
- **clean_dir** (*bool*) – Whether to remove the files in the directory before execution.
- **blocking** (*bool*) – Whether to wait for the run to finish.
- **force** (*str*) – Set what to do if a running executable is found in the directory. Set to "ignore" to launch anyway, possibly resulting in multiple instances running simultaneously; set to "kill" to terminate the existing processes.
- **mpcaller** (MPCaller) – An instance controlling multithreaded calls. If supplied, all calls will be handled by this instance and the blocking parameter will be ignored.
- **create_only** (*bool*) – Whether just to create the files, but not starting the run.

Returns A Run instance describing the execution.

Return type tuple

duat.run.**run_config_grid**(*config*, *run_dir*, *prefix=None*, *run_name='osiris_run'*, *remote_dir=None*, *clean_dir=True*, *prolog=''*, *epilog=''*, *create_only=False*)
Queue a OSIRIS run in a compatible grid (e.g., Sun Grid Engine).

Parameters

- **config** (*ConfigFile*) – The instance describing the configuration file.
- **run_dir** (*str*) – Folder where the run will be carried.
- **prefix** (*str*) – A prefix to run the command. If None, "mpirun -np X " will be used when a config with multiple nodes is provided.
- **run_name** (*str*) – Name of the job in the engine.
- **remote_dir** (*str*) – If provided, a remote directory where the run will be carried, which might be only available in the node selected by the engine. Note that if this option is used, the returned Run instance will not access the remote_dir, but the run_dir. If the remote node is unable to access the path (trying to create it if needed), OSIRIS will be started in the run dir and errors will be logged by the queue system.
- **clean_dir** (*bool*) – Whether to remove the files in the directory before execution.
- **prolog** (*str*) – Shell code to run before calling OSIRIS (but once in the remote_dir if asked for).
- **epilog** (*str*) – Shell code to run after calling OSIRIS.
- **create_only** (*bool*) – Whether just to create the files, but not queueing the run.

Returns A Run instance describing the execution.

Return type *Run*

duat.run.**run_variation**(*config*, *variation*, *run_base*, *caller=None*, *on_existing=None*, ***kwargs*)

Make consecutive calls to `run_config()` with ConfigFiles generated from a variation.

Parameters

- **config** (*ConfigFile*) – Base configuration file.
- **variation** (*Variation*) – Description of the variations to apply.
- **run_base** (*str*) – Path to the directory where the runs will take place, each in a folder named var_number.
- **caller** (int or *MPCaller*) – If supplied, the calls will be managed by a MPCaller instance. If an int is provided an MPCaller with such a number of threads will be created. Provide an instance if interested in further controlling.
- **on_existing** (*str*) – Action to do if a run of the variation exists. Only the names of the subfolders are used for this purpose, which means the run could be different if the variation or the path have changed. Set to "ignore" to leave untouched existing runs or set to "overwrite" to delete the data and run a new instance. Default is like "ignore" but raising a warning.
- ****kwargs** – Keyword arguments to pass to `run_config()`

Returns List with the Run instances in the variation directory.

Return type list of Run

duat.run.**run_variation_grid**(*config*, *variation*, *run_base*, *run_name='os-var_'*, *remote_dir=None*, *on_existing=None*, ***kwargs*)

Make consecutive calls to `run_config_grid()` with ConfigFiles generated from a variation.

Parameters

- **config** (*ConfigFile*) – Base configuration file.
- **variation** (*Variation*) – Description of the variations to apply.
- **run_base** (*str*) – Path to the directory where the runs will take place, each in a folder named var_number.
- **run_name** (*str*) – Prefix to the name to use in the grid system.
- **remote_dir** (*str*) – If provided, a remote directory where the runs will be carried, which might be only available in the node selected by the engine. See `run_config_grid()`.
- **on_existing** (*str*) – Action to do if a run of the variation exists. Only the names of the subfolders are used for this purpose, which means the run could be different if the variation or the path have changed. Set to "ignore" to leave untouched existing runs or set to "overwrite" to delete the data and run a new instance. Default is like "ignore" but raising a warning.
- ****kwargs** – Keyword arguments to pass to `run_config_grid()`.

Returns List with the Run instances in the variation directory.

Return type list of Run

4.3 duat.plot

Plot OSIRIS-generated data.

class duat.plot.**Diagnostic**(*data_path*)

A OSIRIS diagnostic.

data_path
str – Path to the directory were the data is stored.

data_name
str – A friendly name for the data.

units
str – The name of the unit the magnitude is measured in.

dt
float – The time step between snapshots of a consecutive number.

t_0
float – The time of the first snapshot.

time_units
str – The name of the unit of time.

file_list
list of str – List of h5 files, one per time snapshot.

snapshot_list
list of int – List of integers identifying the snapshots. Multiply by dt to get time.

keys
list of str – Names of the datasets in the Diagnostic, given in human order.

axes
list of dict – Info of each axis in the Diagnostic.

datasets_as_axis
dict – Info of datasets if treated as axes. WARNING: Only used with energy bins.

shape
tuple – A tuple with:

- list: The number of grid dimensions.
- int: The number of datasets (excluding axes definition).
- int: The number of snapshots in time.

Note: The axes list is provided in the order of the numpy convention for arrays. This is the opposite of order used to label the axes in the hdf5 files. For example in a 2d array the first axes will be the labeled as AXIS2, and the second will be AXIS1. Unless the user makes use of other external tools to read the data, he/she can safely ignore this note.

__init__(*data_path*)

Create a Diagnostic instance.

Parameters **data_path** (*str*) – Path of the directory containing the diagnostic data.

Raises ValueError – If there is no data in *data_path*.

__repr__()

Return repr(self).

axes_2d_colormap(*output_path=None*, *dataset_selector=None*, *axes_selector=None*, *time_selector=None*, *dpi=200*, *latex_label=True*, *cmap=None*, *norm=None*, *show=True*, *rasterized=True*, *contour_plot=False*)

Generate a colormap in two axes.

A single time snapshot must be selected with the time_selector parameter. For an animated version in time see the `time_2d_animation()` method.

Note: For simple manipulation like labels or title you can make use of the returned tuple or a *matplotlib.pyplot.style.context*. More advanced manipulation can be done extracting the data with the `get_generator()` method instead.

Parameters

- **output_path** (*str*) – The place where the plot is saved. If "" or None, the figure is not saved.
- **dataset_selector** – See `get_generator()` method.
- **axes_selector** – See `get_generator()` method.
- **time_selector** – See `get_generator()` method.
- **dpi** (*int*) – The resolution of the file in dots per inch.
- **latex_label** (*bool*) – Whether for use LaTeX code for the plot.
- **cmap** (str or *matplotlib.colors.Colormap*) – The Colormap to use in the plot.
- **norm** (str or *matplotlib.colors.Normalize*) – How to scale the colormap. For advanced manipulation, use some Normalize subclass, e.g., colors.SymLogNorm(0.01). Automatic scales can be selected with the following strings:
  - "lin": Linear scale from minimum to maximum.
  - "log": Logarithmic scale from minimum to maximum up to vmax/vmin>1E9, otherwise increasing vmin.
- **show** (*bool*) – Whether to show the plot. This is blocking if matplotlib is in non-interactive mode.
- **rasterized** (*bool*) – Whether the map is rasterized. This does not apply to axes, title... Note non-rasterized images with large amount of data exported to PDF might challenging to handle.
- **contour_plot** (*bool*) – Whether contour lines are plot instead of the density map.

Returns Objects representing the generated plot.

Return type (*matplotlib.figure.Figure*, *matplotlib.axes.Axes*)

get_axes(*dataset_selector=None*, *axes_selector=None*)

Get a dictionary with the info of the axes obtained as result of a given reduction.

Parameters

- **dataset_selector** – See `get_generator()` method.
- **axes_selector** – See `get_generator()` method.

Returns Ordered list of the axes left by the reduction.

Return type list of dict

get_generator(*dataset_selector=None*, *axes_selector=None*, *time_selector=None*)
Get a generator providing data from the file.

Calling this method returns a generator which, when called, will provide data for increasing times (unless modified by time_selector parameter). The data might be reduced either by selecting a position in an axis (or a dataset) or by using a function along some axis (or datasets), e.g., a sum.

This data is provided as numpy arrays where the first axis refers to dataset coordinate (if present) and next to (non-reduced) axis in the order they are found in the files.

Parameters

- **dataset_selector** (*str, int or callable*) – Instructions to reduce datasets. An int selects a dataset in human order, a str selects it by name. A function taking a list and returning a scalar can be used to reduce the data, e.g., sum, mean...
- **axes_selector** (*tuple*) – Instructions to reduce axes data. It must be a tuple of the same length of the number axes or None to perform no reduction. Each element can be of the following types:
  - int: Select the item in the given position.
  - None: No reduction is performed in this axis.
  - ScaledFunction: A function applied in a range selected by simulation units.
  - callable (default): Reduce the data along this axes using the given function (e.g., mean, max, sum...).
- **time_selector** (*slice or ScaledSlice*) – A slice or ScaledSlice instance selecting the points in time to take. A slice selects times from the list returned by `get_time_list()`. A ScaledSlice chooses a slice that best represents a choice in terms of time units.

Returns A generator which provides the data.

Return type generator

get_time_list(*time_selector=None*)
Get the list of times obtained as a result of a given slice.

Parameters **time_selector** – See `get_generator()` method.

Returns The times resulting as a consequence of the slice.

Return type `list` of `float`

time_1d_animation(*output_path=None*, *dataset_selector=None*, *axes_selector=None*, *time_selector=None*, *dpi=200*, *fps=1*, *scale_mode='expand'*, *latex_label=True*, *interval=200*)
Generate a plot of 1d data animated in time.

If an output path with a suitable extension is supplied, the method will export it. Available formats are mp4 and gif. The returned objects allow for minimal customization and representation. For example in Jupyter you might use *IPython.display.HTML(animation.to_html5_video())*, where *animation* is the returned *FuncAnimation* instance.

Note: Exporting a high resolution animated gif with many frames might eat your RAM.

Parameters

- **output_path** (*str*) – The place where the plot is saved. If "" or None, the plot is shown in matplotlib.
- **dataset_selector** – See `get_generator()` method.
- **axes_selector** – See `get_generator()` method.
- **time_selector** – See `get_generator()` method.
- **interval** (*float*) – Delay between frames in ms. If exporting to mp4, the fps is used instead to generate the file, although the returned objects do use this value.
- **dpi** (*int*) – The resolution of the frames in dots per inch (only if exporting).
- **fps** (*int*) – The frames per seconds (only if exporting to mp4).
- **scale_mode** (*str*) – How the scale is changed through time. Available methods are:
  - "expand": The y limits increase when needed, but they don't decrease.
  - "adjust_always": Always change the y limits to those of the data.
  - "max": Use the maximum range from the beginning.
- **latex_label** (*bool*) – Whether for use LaTeX code for the plot.

Returns Objects representing the generated plot and its animation.

Return type (*matplotlib.figure.Figure*, *matplotlib.axes.Axes*, *matplotlib.animation.FuncAnimation*)

Raises `FileNotFoundError` – If tried to export to mp4 but ffmpeg is not found in the system.

time_1d_colormap(*output_path=None*, *dataset_selector=None*, *axes_selector=None*, *time_selector=None*, *dpi=200*, *latex_label=True*, *cmap=None*, *norm=None*, *show=True*, *rasterized=True*, *contour_plot=False*)

Generate a colormap in an axis and the time.

This function plots a magnitude depending on ONE spatial coordinate (hence the name) and on time as a colormap in the cartesian product of such a magnitude and the time.

Note: For simple manipulation like labels or title you can make use of the returned tuple or a *matplotlib.pyplot.style.context*. More advanced manipulation can be done extracting the data with the `get_generator()` method instead.

Parameters

- **output_path** (*str*) – The place where the plot is saved. If "" or None, the figure is not saved.
- **dataset_selector** – See `get_generator()` method.
- **axes_selector** – See `get_generator()` method.
- **time_selector** – See `get_generator()` method.
- **dpi** (*int*) – The resolution of the file in dots per inch.
- **latex_label** (*bool*) – Whether for use LaTeX code for the plot.
- **cmap** (str or *matplotlib.colors.Colormap*) – The Colormap to use in the plot.

- **norm** (str or *matplotlib.colors.Normalize*) – How to scale the colormap. For advanced manipulation, use some Normalize subclass, e.g., colors.SymLogNorm(0.01). Automatic scales can be selected with the following strings:
  - "lin": Linear scale from minimum to maximum.
  - "log": Logarithmic scale from minimum to maximum up to vmax/vmin>1E9, otherwise increasing vmin.
- **show** (*bool*) – Whether to show the plot. This is blocking if matplotlib is in non-interactive mode.
- **rasterized** (*bool*) – Whether the map is rasterized. This does not apply to axes, title... Note non-rasterized images with large amount of data exported to PDF might challenging to handle.
- **contour_plot** (*bool*) – Whether contour lines are plot instead of the density map.

Returns Objects representing the generated plot.

Return type (*matplotlib.figure.Figure*, *matplotlib.axes.Axes*)

time_2d_animation(*output_path=None*, *dataset_selector=None*, *axes_selector=None*, *time_selector=None*, *dpi=200*, *fps=1*, *cmap=None*, *norm=None*, *rasterized=True*, *z_min=None*, *z_max=None*, *latex_label=True*, *interval=200*)

Generate a plot of 2d data as a color map which animated in time.

If an output path with a suitable extension is supplied, the method will export it. Available formats are mp4 and gif. The returned objects allow for minimal customization and representation. For example in Jupyter you might use *IPython.display.HTML(animation.to_html5_video())*, where *animation* is the returned *FuncAnimation* instance.

Note: Exporting a high resolution animated gif with many frames might eat your RAM.

Parameters

- **output_path** (*str*) – The place where the plot is saved. If "" or None, the plot is shown in matplotlib.
- **dataset_selector** – See `get_generator()` method.
- **axes_selector** – See `get_generator()` method.
- **time_selector** – See `get_generator()` method.
- **interval** (*float*) – Delay between frames in ms. If exporting to mp4, the fps is used instead to generate the file, although the returned objects do use this value.
- **dpi** (*int*) – The resolution of the frames in dots per inch (only if exporting).
- **fps** (*int*) – The frames per seconds (only if exporting to mp4).
- **latex_label** (*bool*) – Whether for use LaTeX code for the plot.
- **cmap** (str or *matplotlib.colors.Colormap*) – The Colormap to use in the plot.
- **norm** (str or *matplotlib.colors.Normalize*) – How to scale the colormap. For advanced manipulation, use some Normalize subclass, e.g., colors.SymLogNorm(0.01). Automatic scales can be selected with the following strings:
  - "lin": Linear scale from minimum to maximum.

- ”log”: Logarithmic scale from minimum to maximum up to vmax/vmin>1E9, otherwise increasing vmin.

- **rasterized** (*bool*) – Whether the map is rasterized. This does not apply to axes, title... Note non-rasterized images with large amount of data exported to PDF might challenging to handle.

Returns Objects representing the generated plot and its animation.

Return type (*matplotlib.figure.Figure*, *matplotlib.axes.Axes*, *matplotlib.animation.FuncAnimation*)

Raises `FileNotFoundError` – If tried to export to mp4 but ffmpeg is not found in the system.

class duat.plot.**ScaledFunction**(*f*, *start*, *stop*)

A function that applies to a range defined in simulation units (instead of list position).

This object can be used to describe a component of an axes_selector parameter.

__init__(*f*, *start*, *stop*)

Create a ScaledFunction instance.

Parameters

- **f** (*Callable*) – The function to be applied to the range, e.g., a sum.
- **start** (*float*) – Where the application range should start. Actual start will be before if needed to include the given point
- **stop** (*float*) – Where the slice should stop. Actual stop will be after if needed to include the given point.

__repr__()

Return repr(self).

class duat.plot.**ScaledSlice**(*start*, *stop*, *step=None*)

A slice described in simulation units (instead of list position).

This object can be used to describe a time_selector parameter.

__init__(*start*, *stop*, *step=None*)

Create a ScaledSlice instance.

Parameters

- **start** (*float*) – Where the slice should start. Actual start will be before if needed.
- **stop** (*float*) – Where the slice should stop. The point is in general excluded, as usual in Python.
- **step** (*float*) – The desired step of the slice. Actual step will be the biggest multiple of the mesh step which is less than this one.

__repr__()

Return repr(self).

duat.plot.**get_diagnostic_list**(*run_dir='.'*)

Create a list with the diagnostic found in the given run directory.

Parameters **run_dir** (*str*) – The run directory.

Returns List of the diagnostic found.

Return type `list` of *Diagnostic*

4.4 duat.common

Common tools.

class duat.common.**Call**(*fn*, **args*, ***kwargs*)
Objectization of a call to be made. When an instance is called, such a call will be made.

__call__()
Call self as a function.

__init__(*fn*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

class duat.common.**MPCaller**(*num_threads=2*)
MultiProcessing Caller. Makes calls using multiple subprocesses.

processes
list of multiprocessing.Process – Processes managed by the instance.

__init__(*num_threads=2*)
Initialize self. See help(type(self)) for accurate signature.

__repr__()
Return repr(self).

abort(*interrupt=False*)
Remove all queued calls and ask processes to stop.

Parameters **interrupt** – If True, terminate all processes.

add_call(*call*)
Add a call to the instance's stack.

Parameters **call** (*Callable*) – A function whose call method will be invoked by the processes. Consider using lambda functions or a *Call* instance.

spawn_threads(*num_threads*)
Create the required number of processes and add them to the caller.

This does not remove previously created processes.

wait_calls(*blocking=True*, *respawn=False*)
Ask all processes to consume the queue and stop after that.

Parameters

- **blocking** (*bool*) – Whether to block the call, waiting for processes termination.
- **respawn** (*bool*) – If blocking is True, this indicates whether to respawn the threads after the calls finish. If blocking is not True this is ignored (no automatic respawn if non-blocking).

duat.common.**ensure_dir_exists**(*path*)
Ensure a directory exists, creating it if needed.

Parameters **path** (*str*) – The path to the directory.

Raises OSError – An error occurred when creating the directory.

duat.common.**ensure_executable**(*path*, *all_users=None*)
Ensure a file is executable.

Parameters

- **path** (*str*) – the path to the file.

- **all_users** (*bool*) – whether it should be make executable for the user or for all users.

duat.common.**get_dir_size**(*dir_path*)
Get the size of a directory in bytes.

duat.common.**head**(*path*, *lines=10*)
Get the first lines of a file.

Parameters

- **path** (*str*) – Path to the file to read.
- **lines** (*int*) – Number of lines to read.

Returns The lines found.

Return type list of str

duat.common.**human_order_key**(*text*)
Key function to sort in human order.

duat.common.**ifd**(*d*, *v1*, *v2*, *v3*)
Conditionally select a value depending on the given dimension number.

Parameters

- **d** (*int*) – Dimension number (1, 2, or 3).
- **v1** – The value if d = 1.
- **v2** – The value if d = 2.
- **v3** – The value if d = 3.

Returns v1, v2 or v3.

duat.common.**tail**(*path*, *lines=1*, *_step=4098*)
Get the last lines of a file.

Parameters

- **path** (*str*) – Path to the file to read.
- **lines** (*int*) – Number of lines to read.
- **_step** (*int*) – Size of the step used in the search.

Returns The lines found.

Return type list of str

4.5 duat.data

Useful data for PIC simulations.

duat.data.**critical_density**(*wavelength=800*)
Get the critical density for a laser with the given wavelength.

Parameters **wavelength** – Laser wavelength (in nm)

Returns (float) Critical density (particles/cm^3)

duat.data.**density**(*material*)
Get a density (under certain 'normal' conditions) of a material (g/cm^3).

Data is taken from the periodictable package (public domain). The user should check if the returned density fits its application

Parameters **material** (*str or int*) – Name of the material (e.g., 'Al' or 'water') or atomic number.

Returns The density (g/cm^3).

duat.data.**full_ionization_density**(*material*, *z=1*)
Get the full ionization density of a Z (e.g. "Al") when z electrons per atom are ionized.

Parameters

- **material** (*str or int*) – Name of the material (e.g., 'Al' or 'water') or atomic number.
- **z** (*int*) – Number of electrons ionized per atom (a global factor).

Returns (float) Full ionization density (particles/cm^3)

duat.data.**molar_mass**(*material*)
Get the molar mass of a material (g/mol).

Data is taken from the periodictable package (public domain).

Parameters **material** (*str or int*) – Name of the material (e.g., 'Al' or 'water') or atomic number

Returns The molar mass (g/mol).

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

Symbols

A

C

D

E

F

G

H

I

J

K

M

N

O

P

R

S

T

U

V

W

Z